Editorial

A Vision of an Industry

By Carl Helmers

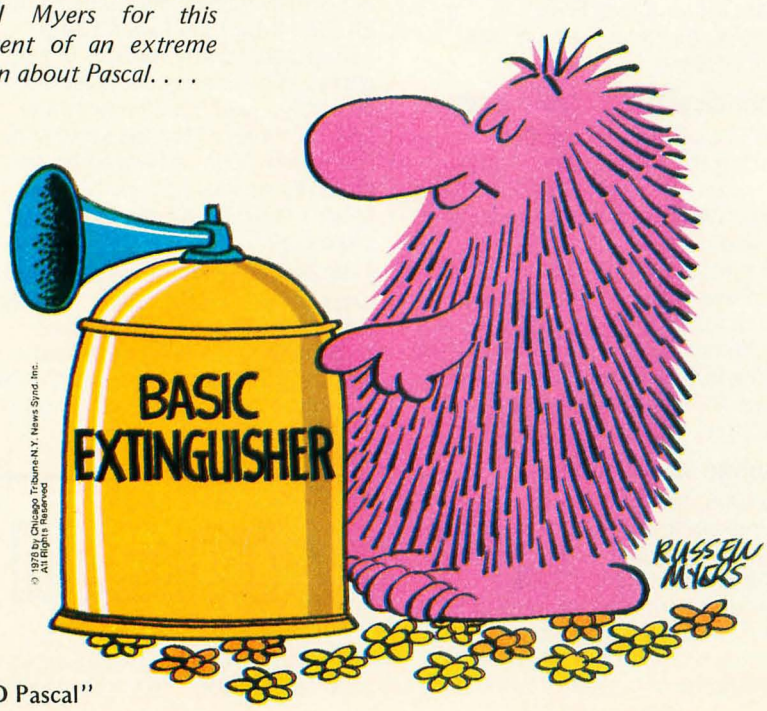
In mid-March of this year, I finished a trip to the West Coast by having a day long meeting with Ken Bowles and his associates at the University of California, San Diego. The purpose of this meeting was to explore some of the possibilities which arise from the standardization of extensions to Niklaus Wirth's language Pascal, and the equally important implications of the technology of intermediate languages such as the optimized form of "P-code" developed at UCSD.

I came to this meeting with a background of familiarity with the reasons for encouraging highly structured languages such as Pascal. Before starting BYTE, I had been involved with the NASA HAL/S language developed by my employer of the time, Intermetrics Inc of Cambridge MA. I lived and breathed considerations of software reliability, ease of program design and the conceptual economy of a detailed program representation which doubles as the documentation of the algorithm. My personal experiences were with the context of the need to "man rate" the flight software of a contemporary spaceship through the use of high reliability software tools and techniques. These points are made elegantly in a number of books and papers which have been published on the subject to date.

What came out of this meeting with Ken Bowles is a vision of an important synthesis of machine independent software representations, the technology of printing machine readable software on paper, and the distribution of software in the form of conventionally printed and bound publications. It is a vision of what the software publishing business could look like over the course of the next few years.

Out of this vision of a machine independent software publishing industry comes a serendipitous justification for support of Ken Bowles' efforts to establish a "bandwagon" effect of support for the Pascal language and machine independent software systems. The purpose of this essay is to discuss the present dimensions of the software publishing problem, the technology which exists for preparing and printing machine readable representations, and the vision of machine independent software publishing which Ken Bowles and I saw inherent in the Pascal P-code technology as we discussed it that day.

Based on a computer graphic suggestion by students Joel McCormack and Owen Hampton at UCSD, we arranged with Russell Myers for this statement of an extreme opinion about Pascal. . . .



"UCSD Pascal"

Publishing Software

As the users of the personal computer expand in number, the means of distribution of software become critical to those who would distribute such software. In personal computing we are faced with a kind of problem which is completely new in the computer industries: the number of machines installed is becoming incredibly large by standards of the past 20 years, and the price paid per unit installation is becoming incredibly small. The computers which are a potential market for software are in the initial stages of becoming a mass market: too large a market for the custom craftsmanship of the traditional software vendor. To be convenient for the customers programs must be distributed with a machine readable copy which eliminates the need for hand key-

Continued on page 133

Continued from page 6

stroking of programs or object code for programs. The traditional manual and job shop methods of production of copies of software for distribution are not appropriate when we think of a mass market of 10,000 to 100,000 copies (or more?) of a program distributed via retailers and mail order houses with a retail price of (for example) $9.95.

The Software Distribution Model

Given an identifiable set of computers with sufficiently similar characteristics, software can be marketed and distributed to multiple users.

The "sufficiently similar" characteristics which make a program marketable to multiple users include the formal representation of the software, and the machine readable medium in which the software is delivered. The machine readable representation of a program product is always accompanied on delivery by extensive printed documentation. At a minimum this documentation describes how to use the product; in the optimal case it includes details of the actual algorithms employed. To summarize, the key points of a delivered product are:

- Formal representation.
- Machine readable medium.
- Documentation.

I'll be making evaluations and comments largely on the subject of formal representation from the point of view of the new mass market for software which is developing in the personal computing field.

Formal Representation

The formal representation of programs to be distributed by a software vendor is one of the key choices which has to be made. At one extreme, the vendor could provide extremely machine dependent and configuration dependent low level code for a particular computer system product. At the other extreme, the vendor of software might provide a largely machine independent formal representation in a high level language shared by a number of computers. At an intermediate point between these extremes, especially in an era of mass production of a small number of processor architectures as microcomputer systems, we find the possibility of delivering configuration independent but machine dependent relocatable representations of low level code for a particular microprocessor instruction set.

For that class of software products supplied by the original manufacturer of a particular computer system, there is no problem providing compatible software at whatever level of representation is chosen. The manufacturer of a system after all controls the detail choices with respect to processor hardware, system configuration and systems software. Since all the details are decided by the particular design, it is even practical to market software in the form of a memory image at the lowest level (possibly in read only memory parts). Since the choice of processor is well defined, the manufacturer can also provide modules of software represented as relocatable machine code, along with a suitable loader program which is part of his systems software. Since the detailed choice of high level language processors is well defined, the manufacturer can also provide applications and systems programs represented in *his* or *her* high level language. The manufacturer of computer systems products at most must deal with a small integer number of processors and high level languages.

We find this model of software delivery by the manufacturer of a system throughout the computer industry to date. Every mainframe and minicomputer comes with low level representations of systems software and (eventually, if not at introduction) with user

Articles Policy

BYTE Publications Inc is continually seeking quality manuscripts written by individuals who are applying personal computer systems, designing such systems, or who have knowledge which will prove useful to our readers. For a more informal description of procedures and requirements, potential authors should send a self-addressed, stamped envelope to BYTE Authors' Guide, 70 Main St, Peterborough NH 03458.

Articles which are accepted are purchased with a rate of $45 per published page, based on technical quality and suitability for the intended readership. As to articles appearing in BYTE magazine, each month, the authors of the two leading articles in the reader poll (BYTE's Ongoing Monitor Box or "BOMB") are presented with bonus checks of $100 and $50. Unsolicited materials should be accompanied by full name and address, as well as return postage.■

libraries of high level and low level programs applicable with the particular systems. At the lowest end of the personal computer spectrum of functions we find a similar case: the major programmable calculator manufacturers with their independent incompatible systems provide users with libraries of magnetic cards or read only memories expressed in a form consistent with the particular machines.

But a characteristic of manufacturers of computers is already evident again in the personal computer world, just as it previously existed in the world of minicomputers and larger computers: whatever the resources of the manufacturer, there is no way it can cover all the myriad applications possible for its computer. To draw an analogy from music, we hardly expect a piano or organ company to supply sheet music ("software") with the musical instrument which is suitable for every user's tastes. The music "user" purchases scores according to personal likes. A personal computer provides an analogous opportunity to exercise tastes in software characteristics. Even for the traditional high priced computer, customization through software is for the most part independent of the manufacturer once the basic operating system and software tools have been defined.

In software, the past has seen a large number of custom software vendors grow large in the niches of large scale computing and minicomputer technology. As the number of people using personal computer systems increases due to the low price of these systems, independent software publishing seems to be one of the most promising ways to assure a wealth of options to the user, provided that the difficulties of the N-representation problem can be overcome.

The N-Representation Problem

For the moment, let's ignore all reference to the problem of machine readable data compatibility and simply look at the user's point of view with respect to software. The user has purchased computer X for use in personal or professional contexts. When he or she has made the commitment to the system, our user can in general expect to be able to conveniently load programs created on other X systems from the same manufacturer. But what if he or she wants to load a program created by a neighbor on computer Y from another manufacturer? Or if the user wants to load a program from an independent software vendor? The variety of representations available in the traditional world of computers as well as the personal computer world is large — even within the framework of nominally machine independent high level languages.

Confining ourselves just to machine

dependent microcomputer assembly languages, there is a wide choice of architectures. At present we find the 8080, Z-80, 6502 and 6800 dominate personal computer architectures. Over the next two to three years we will find added to this list the 9900, 8086, Z-8000 and 6809. If the user of a personal computer sees a neat application system which only comes represented in 8080 code when he has a 6800, that user is effectively unable to run it without a recoding effort. (But even confining ourselves to assembly languages of the same machine design, there is often incompatibility. One vendor of Z-80 software provided an assembler using a hybrid extension of 8080 mnemonics, while others use Zilog Z-80 mnemonics. So the same processor has at least two low level languages available.)

Turning to high level languages, the machine independence of software becomes much greater. But current practices in the personal computing industry are far from machine independent. There is a de facto standard BASIC interpreter in existence, available on most 6502 and 8080 or Z-80 systems. This standard high level language is that defined by the Microsoft company. Extensions and changes of detail accompany each implementation, especially when a given computer has specialized graphics capabilities not available on all the other computers. With the Microsoft design, the major portions of an extended BASIC are identical over a large set of machines.

But Microsoft BASIC is not the only interpreter in existence. A very prominent BASIC in terms of the number of users employing it as represented in the unsolicited articles received at BYTE is the North Star BASIC interpreter. This interpreter is widely used on 8080 and Z-80 systems because of the wide availability of the small floppy disk systems manufactured by that firm: buying a North Star disk peripheral for an S-100 bus system gets the user a limited operating system and the North Star BASIC. The North Star BASIC interpreter and the Microsoft interpreter are inconsistent in a number of fundamental ways in areas of string handling and array dimensions. And these are but the two most prominent interpreters as seen from my point of view as editor of BYTE. I could almost comment that manufacturers take any random formulation of a language vaguely resembling BASIC as originally implemented at Dartmouth, and call it BASIC for marketing reasons. (The temptation to add or delete "features" in a language is of course not confined to BASIC alone.)

From the point of view of a software publisher, the economies of scale obtainable from a mass market will only be obtained if we use a common representation for applica-

tions and systems programs which can be correctly executed by any low level architecture available in the marketplace. With a large number of mutually incompatible software systems, this is not the case. It is my contention that the N-representation problem can be solved *once* by use of appropriate intermediate language representation and efficient interpreters for particular microprocessors. Then the key part of an application or systems program product is the high level language documentation, the equivalent lower level intermediate language object code, and the user documentation: all of course independent of the final machine upon which the software will run. The only machine dependent part which needs to be published is the intermediate language interpreter for a given machine and system configuration. This machine dependent part needs only one definition and one publication version.

Given an interpreter definition, the standard high level language, and the standard intermediate language representation of programs, the user can be assured that once the object code is in place in his machine, the program will run with the same characteristics as described in the documentation for a radically different machine. (Hardware differences due to favorable number representations will make differences in precision and accumulated numeric error effects of course.)

Ruling Out BASIC

To the software publisher, a choice of a high level language and intermediate representation for executable code presents a moderate problem. The widely used BASIC interpreters could be used for a perfectly functional representation for the code of many programs. But such interpreters suffer from many inherent disadvantages:

- Lack of uniform representation.
- Slowness of execution.
- Archaic nature of BASIC.
- Lack of a compact machine independent compiled form.

I've already commented on the lack of uniformity in the various BASIC implementations. The slowness of execution is inherent in this type of interpreter. In extreme cases an active search through memory for a label op code is used to find targets of subroutine calls or unconditional transfers. At best there is a level of semantic interpretation necessary to convert a condensed version of the source code into executed code. Many applications and systems programs cannot tolerate the lack of speed inherent in such interpreters. But BASIC can be *compiled* instead of *interpreted*, so this argument alone is far from sufficient to rule out BASIC.

More important, a language like BASIC as presently implemented reflects an earlier state in the evolution of computer languages, circa the early 1960s, with innumerable ad hoc patches and fixups to add "features." Through the 1960s and early 1970s advances were made in the concept of what a computer language should be in order to be convenient to use and conducive to error free thinking and programming. (For just one contrast, consider this: where the BASIC programmer is required to go almost to the machine language level of assigning numbers to locations in a program, good

contemporary high level languages such as Pascal and its relatives allow the programmer to use meaningful names based on the application being programmed.)

Finally, BASIC as implemented in most cases suffers from the lack of a compact externally available machine independent version of the compiled form of a program. This is an important requirement for the software publisher, since executable code must always be supplied in some machine readable representation, and compactness of representation is important if the inconvenience of relatively slow input techniques is not to discourage the user.

For the reasons just summarized, BASIC is not the ultimate form in which programs are best published. But if BASIC is not the personal computing representation which minimizes the N-representation problem, then what is a better choice?

Enter Pascal

My own personal interest in Pascal came about for reasons which I summarized in the December 1977 BYTE, page 6, in an essay entitled "Is Pascal the Next BASIC?" In this issue several excellent articles including those by Ken Bowles, Chip Weems and Allan Schwartz provide further rationale by way of tutorial argument and example.

This personal viewpoint with respect to Pascal is that of a *user* of a personal computer system who wants to *conveniently* and quickly implement applications and systems software projects ranging from the sublime to the ridiculous. In the sublime category, I include systems software as an art form in itself. I also include writing systems software for my pet projects in musical applications of computers, sophisticated games, and some experiments in the exploration of artificial intelligence concepts. In the ridiculous category, I include such mundane tasks as trivial games, income tax calculations, personal mailing lists of friends and relations, etc. The point about Pascal to be made here is that it is a language well adapted to the utility of computing, whatever your personal definition of utility is. In the range of applications I expect that the Pascal approach to structured, self-documenting, machine independent code will suffice with only an extremely rare necessity to resort to ad hoc kluges in the name of time or memory space efficiency.

From general reading I knew that a Pascal compiler was available and easily transferable to new machines through the use of the technique of "P-code" intermediate language representations. This availability throughout the academic world was one of the reasons for the spread of Pascal, for it is one thing to extemporize about the virtues of a representation and another thing to be able to actually write and examine the properties of code in that representation. Since the original Pascal compilers from Jensen and Wirth et al in Zurich were written in Pascal, producing a P-code intermediate language output file, the task of making the compiler run on a totally new machine architecture was reduced to a relatively simple task of writing an emulator for the hypothetical "P-machine" which executes "P-code" as its machine language.

What I did not know at the time of my earlier comments in these pages is the extent to which that P-code technology had already been applied to small computer systems, in

particular through the work of the people at the University of California at San Diego (UCSD). The UCSD Pascal project has created a nearly machine independent low cost operating system which includes Pascal as the principal high level language, all the usual disk filing system features, support of high resolution bit map graphics including user definable font storage for the character set, an advanced cursor oriented text editor, and interactive compilation and editing features. All the systems software in this package is written in Pascal with the exception of the P-code interpreter and associated detail hooks to the hardware.

The hardware dependent core has already been implemented and is readily available for LSI-11, 8080, Z-80 and 8085 processors. (The cost is only $200 for individual orders, with UCSD quoting a $10 royalty per copy to manufacturers distributing systems in the highest volumes.) At this writing, in the small computer arena, three systems are available which come with UCSD Pascal as a key feature: an LSI-11 system packaged by Terak Corp and heavily used at UCSD, an 8085 processor in a elegant wood finish package with dual floppy drives manufactured by Northwest Microcomputer Systems, and a compact Z-80 system with dual floppy disks manufactured by Altos Computer Systems. Individual users who have 8080 floppy disk systems with the CP/M operating system and enough main memory get a floppy disk to bootstrap UCSD Pascal.

A Serendipitous Result

The nature of the implementation of Pascal compilers, and the UCSD Pascal in particular, leads to an important byproduct: by simply using the UCSD Pascal compiler as the mode of expression of applications programs to be published, it is possible to provide a compact, machine independent representation of programs which greatly simplifies the N-representation problem for the independent software distribution house. The intent of discussing this serendipitous result in print at all is to show the way in which such independent software houses can indeed solve one of the thornier issues and provide their customers with programs which are compiled once yet will run on any one of a number of personal computer systems.

What do we have which already exists in a form which can be readily adapted to a number of small computers? We have the work at UCSD which has produced P-code interpreter based systems for LSI-11 and the family of microprocessors inspired by the 8080 (8080, 8085, Z-80). By the end of the

summer of 1978, indications are that UCSD will also have bootstrapped the Pascal compiler to run on 6502 and 6800 architectures. Taking this P-code interpreter as the input, it is not that difficult to conceive of a self-contained software system which will run in a 16 K byte or larger personal computer system and will contain the necessary interactive user interfaces to load and run a program expressed in the P-code intermediate form as output from the Pascal compiler, but without the necessity of having the full UCSD system available locally to support local compilations.

As a means of demonstrating this concept, a student at UCSD will spend some time this summer creating and characterizing a system based on the UCSD P-code interpreter software for two different machines. This stand alone system will run in the typical current memory sizes of 16 K to 24 K found in personal computers. The goal is to demonstrate a system which can read in a P-code object file (possibly in bar code or audio format), then execute the object file. Issues to be addressed are those of designing the details of the program so that its machine dependent parts can be relocated easily, and so that initial patches for input/output (IO) conventions can be created without excessive mental effort. The machine independent part of this stand alone operating system will be written in Pascal.

In principle, expanding this work to a greater number of processors, it is possible to create a set of Pascal P-code machine emulators which can be published once and only once for each common machine architecture and personal computer manufacturer's configuration, so that this "virtual machine" can be used by a whole family of independent software vendors as a target machine for their wares, rather than requiring each software vendor to solve the N-machine problem separately. By inexpensively publishing the code of the P-machine emulators, we hope to help kindle both an interest in Pascal as a source language and a chain reaction of simplification in the software conventions which must be addressed by independent software vendors. Only time will tell whether or not we accomplish this goal.

A Solution to the N-Machine Problem

Given the existence of such inexpensive standard emulators for the P-machine which executes P-code, a number of beautiful effects become evident for the distribution of application and systems software among a large number of users.

First, since P-code is conducive to use of Pascal as a source language, there will be

a trend toward use of Pascal to express algorithms — a result which is laudable on the abstract and practical grounds of Pascal's beauty as a self-documenting and structured representation of programs. (We already see this trend with respect to BYTE articles presently queued for publication in the near future.)

Second, the N-machine problem of distribution is solved by the device of using the P-machine emulator for each of N-machines as the only machine-specific program, and widely publishing the emulators at as low a cost as possible.

Third, the P-code object code form is a semantically compact representation which in fact minimizes the number of bits necessary to communicate a program to the system which the end user employs. (Yet it maps directly into the source code expressed in Pascal as part of the documentation of the program product in place of flowcharts or other devices.) This consideration is important to the relatively slow IO devices such as FM subcarrier broadcasts of programs, printed bar code copy of programs, audio channel recording of programs, phone network transmission of programs, or silicon real estate of read only memory parts (as inspired by Texas Instruments' SR-59 "Solid State Software" and hinted at by every other semiconductor manufacturer interested in distributing computers at retail).

Why Not Publish Machine Readable Source Code Instead of an Intermediate Language Representation?

The intention of this argument is to provide a way for *compiled* code to be distributed for use with systems which have diverse microprocessor architectures and detail implementations. A key to publishing software inexpensively is the requirement that every detail copy of the software published be identical. Further, a certain definition of the "lowest common denominator" of the set of systems is required.

One way of publishing which is guaranteed to be amenable to a wide variety of detail representations is to publish the machine readable source code of software. But the sheer volume of the code for a well documented source listing argues for a way which is more economical of the user's time and energy. By publishing the machine readable but machine independent intermediate language object code compiled from a printed source listing (also part of a product), the executable representation can be loaded into the machine much more quickly; for program representations in read only memory which are mass produced, an intermediate code representation is also favorable because of compactness relative to source code.

To summarize, the intermediate language approach provides the benefits of *machine independence* coupled with the compactness of representation inherent in the usually *machine dependent* object code for a particular architecture. (The negative side of using a machine independent representation is of course the time overhead of the required low level interpreter. But for a well done intermediate language interpreter, we would expect this penalty could approach a mere 2:1 versus a typical 20:1 or worse penalty for direct interpretation of the source code.)■

Some Notes About Pascal. . .

As this issue was being prepared, a number of interesting bits of information became available:

- *Ken Bowles reports that one associate of the UCSD Pascal project is using the* microcomputer *based Pascal which the project has created in order to write a P-code optimizer in Pascal. The writing of an optimizer program is not in itself particularly noteworthy, but the fact that this optimizer is being written for Pascal compiler output of a Cray-1 computer shows ample evidence of the relative machine independence of Pascal techniques. Here we find the LSI-11 based Terak machines at UCSD (typically a fully loaded LSI-11 with keyboard, bit map graphics, one floppy drive) being used to write, debug and check out programs for one of the world's largest and fastest computers, the Cray-1. (How fast? Fast enough so that light speed propagation limits in the wires become a nontrivial consideration in the physical design of the machine.) Yet the Cray-1 uses a dialect of Pascal for systems programming, and even has a FORTRAN compiler which uses P-code as its intermediate language.*

- *We note that even the US Defense Department likes Pascal as a replacement for such monstrosities as JOVIAL. Two contracts for further language design efforts on the "Steelman" phase of the search for a "DOD-1" language definition have just been announced, with Intermetrics Inc and Honeywell-Bull being finalists in a language design competition based on preliminary proposals. Much of the content of this language definition is expected to be inspired by Pascal, even if it is not a proper superset of the language.*

- *From the industrial side, Texas Instruments Inc has a version of Pascal which is supported for the 990 series of minicomputers, where "supported" means that it is available for use with their disk systems, marketing people are pushing it at seminars for 990 system users, and a comprehensive manual describing the system is available. The 990 series of minicomputers of course includes the microcomputer version of the processor, which is the TMS-9900, and is one of the logical choices for a serious homebrewer or designer of a custom microcomputer system which must use a fair amount of complicated software. The 990 version of Pascal is probably a little too expensive for the individual to purchase, but it represents a very good investment for a commercial user.*

- *Finally, as we went to press with this issue in mid-May, a standards conference, called by Ken Bowles, was scheduled for mid-July at San Diego. Attendance was expected from the worldwide Pascal community, as well as representatives of major industrial concerns, with the intent of defining a set of "standard" extensions to the Pascal language of the Jensen-Wirth report. We expect to have some comments in a future issue about the major points covered in that standards conference. (Of course, the reason for standards must be properly understood: a language standard provides a reference so that any implementer can flag users about how his particular system deviates from the standard. This philosophy is seen throughout computer technology in areas as diverse as character sets for terminals and FORTRAN IV compilers which use the ANSI standard model. A Pascal standards consensus already exists in the Jensen-Wirth report published by Springer-Verlag, and the purpose of the conference is to define an extensions set that covers the superset of the original language necessary to enhance the practicality of the language in real world situations.)*

Pascal is one of the most exciting developments with respect to personal computing we have seen in recent years. The small computer is finally getting to a point where the professionally oriented individual can afford (at the price of a typical new automobile) a computer with some of the most advanced software development characteristics possible in today's computers. Just as a crank starter can get the engine going on an automobile, BASIC and assembly language can indeed be used to program computers. But if one really wants to use an automobile conveniently, an ignition switch and electric starter are now considered essential. The moral of this little simile is that Pascal is the electric starter of the computer world.■